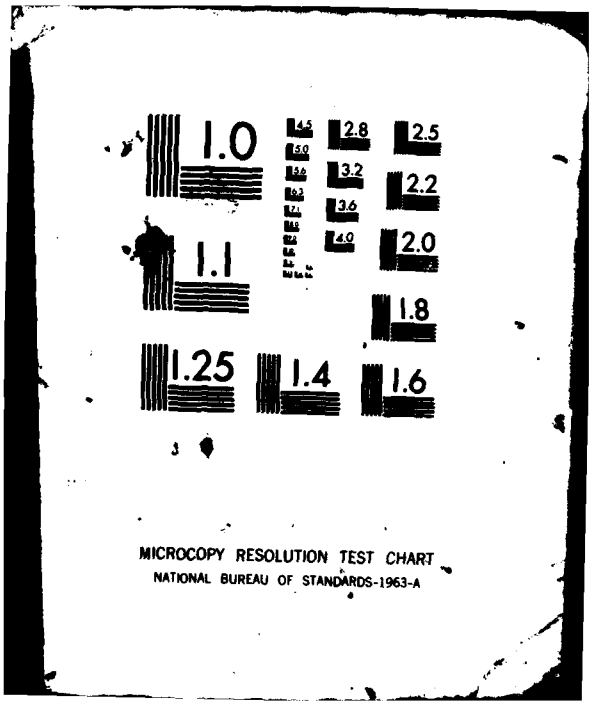
AD-A114 604 ROYAL SIGNALS AND RADAR ESTABLISHMENT MALVERN (ENGLAND) F/G 9/2
ADA ON MULTIPLE PROCESSORS,(U)
MAR 82 J A MCDERMID
UNCLASSIFIED RSRE-MEMO-3464 DRIC-BR-83066 NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Memorandum 3464

TITLE: ADA ON MULTIPLE PROCESSORS

AUTHOR: John A McDermid

DATE: 11 March 1982

SUMMARY

This paper considers a number of possible ways of implementing the Ada rendezvous in a computer system comprising a number of processors. It shows that, in principle, a two phase protocol requiring four messages to be passed is necessary to implement the rendezvous correctly when timed calls are used. However in many cases a simpler, one phase, protocol requiring only two messages per rendezvous can be used.

A comparison is made between the rendezvous and message based communication for situations where one to many, rather than one to one, communication is required. The rendezvous is shown to be very inefficient for implementing one to many communication. Finally some of the problems of loading and running Ada programs are briefly considered.

A

1) Introduction

The purpose of this document is to discuss the problems of implementing Ada on a Multiple Processor Computer System and to compare possible ways of overcoming these problems. We shall be concerned primarily with how a rendezvous can be achieved between tasks running on different computers, as this seems to be the main technical problem. We will compare the different ways of implementing the rendezvous in terms of the number of inter - computer messages, and context switches which they require as these are (probably) the two most important aspects of the overheads involved in providing the rendezvous.

2) Assumptions

We make the following basic assumptions about the characteristics of the Multiple Processor System (MPS) on which we wish to implement Ada. First, it seems that the implementation of the rendezvous in a system having shared main memory can be achieved simply by extending a single processor implementation. It is less obvious how to implement the rendezvous without shared main memory, so we will concentrate our attention on such Multi Computer Systems (MCS). As a consequence we will assume that parameters and results are passed by value rather than by reference.

Second, we assume that there is a kernel running in each computer which supports the Ada tasks, performs scheduling and provides the mechanisms for inter - computer communication. We assume that the kernel need not be written in Ada.

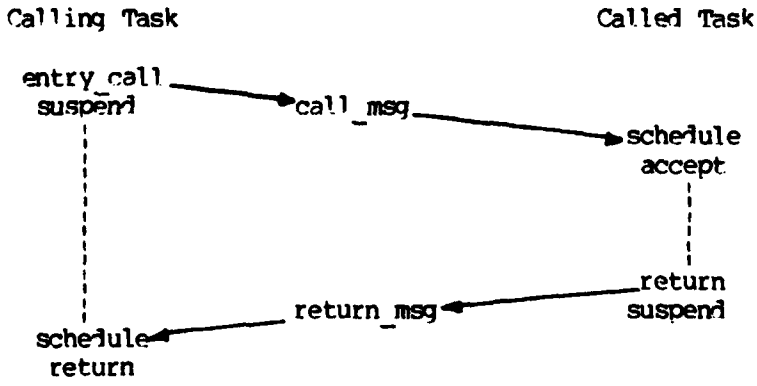
Third, we assume that the communication system provides the abstraction of total reliability, so we will not consider the problems of lost messages etc. Implicitly this means that the transmission of a single message, at the user program level, may require many low level messages to be transmitted. We will always quote the number of messages required by a particular implementation of the rendezvous at the user program level as this gives the simplest basis for comparison.

Finally, we distinguish two different types of context switch. A Task Context Switch (TCS) occurs when a task is scheduled or suspended for some reason (e.g. waiting for an entry call to be accepted). An Interrupt Context Switch (ICS) occurs when an interrupt handler is entered or exited. We will assume that an ICS only occurs twice per received message although this may be very far from the truth with word or byte oriented communication systems. One might expect that the time required to perform an ICS will be smaller than that required to perform a TCS, although this will not universally be true.

It is perhaps worth noting that the Nassi - Habermann optimisation cannot be performed when the communicating tasks are in different computers. However it might be possible to exploit this optimisation if shared main memory is available.

3) Simple Rendezvous

By a simple rendezvous we mean one where the calling task calls the entry unconditionally (and without timeouts), and the called task unconditionally accepts the call (although not necessarily immediately it is issued). Naturally we assume that the calling and called tasks are running on separate computers. The simple rendezvous can be implemented as indicated below:



This straightforward implementation requires two messages to be passed between the computers, and requires two ICS in each machine to handle the receipt of the messages. In the calling task two TCS are required. In the called task up to two TCS may be required, but the number depends on whether or not the task is active immediately before and after the rendezvous. This comment applies throughout the following discussion.

If necessary exceptions can be returned to the calling task: e.g. Tasking_Error can be returned if the called task has terminated.

4) Conditional Entry Calls

These can be implemented in the same basic way as the simple rendezvous, except that a rejection may have to be returned, rather than the result of executing the entry procedure. It is possible that two TCSs will be required in the called task even if the call is rejected. These context switches can be avoided if the kernel (interrupt handling routines) can preserve enough information about the called task to know whether or not a rendezvous is possible, without entering the environment of the called task.

5) Selective Waits

Selective waits can be implemented using the basic method described above.

6) Timed Entry Calls

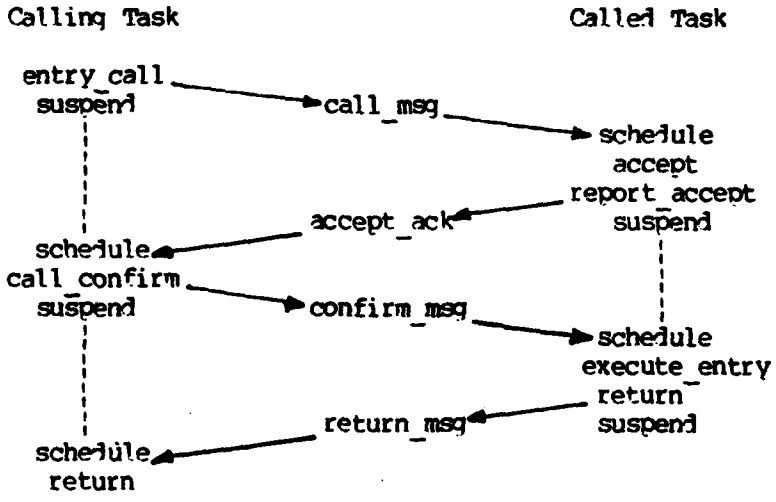
6.1) Introduction

If timed entry calls were implemented by the above mechanism it is possible that the timeout might expire, causing the calling task to execute its alternative code, whilst the called task was executing the entry. This eventuality would be a violation of the rendezvous semantics, and clearly must be avoided.

For a timed entry call, the timing is performed on acceptance of the entry call, not on the execution of the entry. This means that (in principle at least) information needs to be passed back to the calling task once the call is accepted, as well as on the completion of the call. This is the basis of our first alternative solution below.

6.2) Simple Approach

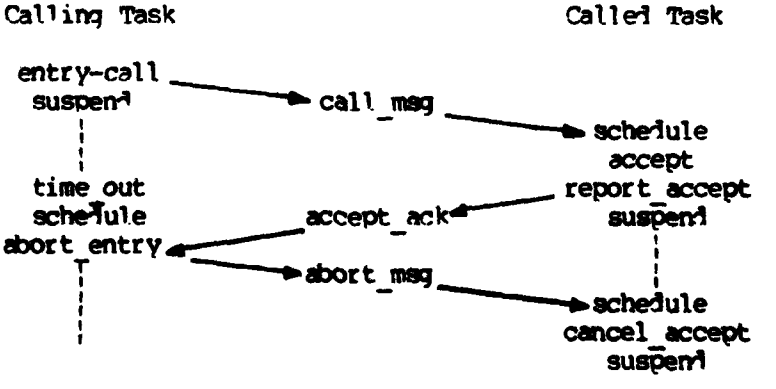
The simplest approach to implementing timed entry calls seems to be to use a two phase "handshake" protocol. The first phase governs the acceptance of the call, and the second is equivalent to the protocol described in section 3 controlling the execution of the entry. The execution of the protocol would be as follows if the call were successful:



This requires four messages, four TCS for each task, and four ICS for each task.

Because of the timeout it is possible for the called task to accept the entry call after the callers timeout has expired, and the task has continued execution. There are two ways of recovering from this situation: the calling task can be "rolled back", so that the entry is executed; or the acceptance of the call can be cancelled. The former course may be very difficult, especially if the calling task has entered into communication with other tasks, or performed some I/O before the accept message is received. The latter course only requires the information recording the acceptance of the call to be changed (as the called task will be suspended). Clearly the latter approach is much simpler to achieve and it correctly implements the semantics of the timed call.

The behaviour will be as follows for a call which is not accepted before the timeout expires:



This requires three messages, two TCS in the calling task, and four in the called task. There will be two ICS in the calling task, and four in the called task.

A problem which arises with this implementation is dealing with accept messages which arrive long after the timeout has expired, and the calling task has continued with its alternative action. There seem to be two possible approaches. The kernel could maintain information regarding the incomplete rendezvous until the accept message is received and the abort message can be generated. This technique may give problems with storage management if large numbers of rendezvous have to be "remembered". Alternatively, the incomplete rendezvous can be forgotten and the kernel can respond to any unrecognised accept message with an abort message. This latter technique will not impose storage overheads but it has the disadvantage that it will not help in the detection of certain error conditions, such as trying to communicate with a task that has already terminated.

Clearly this straightforward implementation is quite costly in terms of messages and context switches. There are, however, more efficient ways of implementing the rendezvous which may be applicable in some circumstances. These methods rely on being able to execute the timeout in the called task.

6.3) Timing at Both Ends

When calling the entry, the calling task could transmit the timeout duration to the called task. Assuming that both tasks have access to clocks running at roughly similar rates, then the called task can inspect the timeout period and not reply if it knows that it will not be able to accept the call quickly enough. Unfortunately we cannot guarantee to avoid the situation where an accept message is received after the callers timeout has expired (unless the timing of the communication etc. is deterministic and well known) so we must still cater for the possibility of having to abort the rendezvous.

6.4) Timing only at Called End

If the message delay through the communication system is well known, deterministic, and short with respect to the timeout period then it may be possible for the called task to execute the timeout on behalf of the calling task. If this is the case then we can use a single phase protocol for performing the rendezvous as we described in section 3. The only change to the protocol of section 3 is that the called task can return a "timed out" message to the caller in order to allow it to continue without performing the rendezvous.

This implementation is attractive in that it is comparatively efficient. However there is a problem to do with reliability. If the processor running the called task fails then the calling task may never (or very belatedly) receive its timeout message. One of the reasons for using timed entry calls may be to allow detection of, and recovery from, remote failures, in which case timing at the called end may not be satisfactory. Arguably it should be the responsibility of the kernel to detect remote failures and to return a suitable exception to the task. However the time taken by the kernel to detect the failure may be much longer than the time the calling task is willing to wait. It seems therefore that there will be circumstances under which the two phase protocol will have to be used.

7) Summary of Rendezvous Implementations

We have described a number of ways in which the rendezvous can be implemented. The most efficient method which we can use is a simple, one

phase protocol as described in section 6.4. This protocol can cater for the simple rendezvous, conditional rendezvous and, under some circumstances, with timed rendezvous. This protocol will typically require two messages, two TCS per task, and two ICS per task.

For the circumstances where the one phase protocol is not acceptable, e.g. where we wish to recover from the failure of remote computers, then the two phase protocol described in section 6.2 will have to be used. This protocol has twice the overhead of the single phase protocol if the rendezvous is completed successfully. If the rendezvous is not completed (due to a timeout expiring) then the overheads are rather lower.

8) Comparison with Message Passing

8.1) Message Passing Paradigms

There are essentially three distinct forms of message passing inter process communication scheme. The simplest simply consists of the sender transmitting a message, and continuing execution without an acknowledgement ever being returned. This scheme does not make it easy to detect remote failures or lost messages, but may be quite appropriate under certain circumstances - e.g. transmission of data from a sensor, where the loss of the occassional reading will not adversely affect the behaviour of the system.

The second paradigm is that of one process sending a message, then waiting until the message is acknowledged by the process which received the message. This method gives scope for detecting and recovering from certain classes of errors (e.g. lost messages caused by communications failures), and clearly matches the semantics of the rendezvous.

The third paradigm is that of sending a message and receiving an explicit acknowledgement, with the sending process able to continue processing between sending the message and receiving the acknowledgement. The advantage this offers over the second method is that concurrency is improved since the sending process may be performing useful work whilst waiting for an acknowledgement. This form of behaviour can only be achieved in Ada by the artifice of creating a task specifically for performing the inter - task communication, thus allowing the parent task to continue executing. This technique can have severe disadvantages as described below.

8.2) The Two Phase Commit Protocol

We describe the implementation of the Two Phase Commit Protocol in Ada as it serves to show the problems which arise from the fact that a task cannot continue executing between sending a message and receiving an acknowledgement (i.e. the task is suspended whilst the rendezvous is in progress). We believe this protocol to be a very salient example as it is widely used as a way of ensuring consistency control in distributed database systems.

We can adequately describe the most important features of the Two Phase Commit Protocol (2PC) by the following example. Imagine that we have a replicated database with a total of N copies, and we wish to update the database so that all the copies remain in step. In essence we need to update all the copies indivisibly. This is achieved by one task (the control task) notifying all the copies that an update is to be performed, and the tasks responsible for the copies either acknowledge that they can perform the update, or say that the update has to be aborted because it conflicts with

some update already in progress. This is the first, or notification, phase. The originating task then either informs all the cooperating tasks that the update must be aborted, or instructs them to perform the update, as appropriate. In the latter case the tasks will update their copies of the database, then return acknowledgements to the originating task. This is the second, or update, phase.

Clearly the implementation of the protocol will be complicated by the need to deal with failures of remote computers etc., but the basic form is not affected by these considerations. Using our third paradigm for the message passing model, the 2PC control task can transmit messages to all N tasks, then wait for acknowledgements, in both the first and second phases. Thus the message passing discipline allows us to achieve a high degree of concurrency.

If we implemented the 2PC protocol in Ada the simplest way would be to perform the communication with each of the N cooperating tasks in turn, thus having N rendezvous in series in the first phase, and similarly for the second phase. Note that this already requires twice as many messages and context switches as the message based implementation. This implementation will obviously be slow as the inherent parallelism is lost.

In order to improve parallelism we could implement the protocol so that the controlling task spawned N subtasks to perform the communication. This has the undesirable side effect of increasing the number of TCS as control passes between the main and the subtasks. If separate subtasks were used for each phase, and task/subtask synchronisation can be achieved by use of shared variables, then this requires at least another 8N TCS, making a total of 16N TCS, four times the number required using message passing. It will be quite awkward determining when each phase has finished - perhaps the easiest way being to use the subtask statusses to determine when they have all terminated.

It might be possible to improve on these overheads by creating N subtasks for the duration of the 2PC operation, however this would mean that we would have to use rendezvous between the subtasks and the control task in order to signal the end of each phase. This still requires 8N TCS, but would involve less process creation and destruction which must themselves be expensive operations. However it is possible that some of these contexts switches can be eliminated by performing optimisations, analogous to the Nassi - Habermann optimisations.

Some MCS communication systems provide broadcast facilities which can provide very efficient 1:many communication. The Ada rendezvous does not allow such hardware to be exploited.

In short it seems that Ada will enforce very expensive, and quite complex implementations of protocols of the above form. Since this form of protocol will be at the heart of any Distributed Database Manager, and of many other MCS applications, Ada may be a very poor choice of implementation language from the point of view of efficiency, and simplicity of implementation.

9) Program Loading and Hardware Mapping

The mapping of the Ada program onto the available hardware must be specified at some stage in the program development and loading process, unless the mapping is to be chosen automatically by the programming support environment. It should be fairly straightforward to specify the mapping

either as pragmata in the program source text, or as commands to the program loading system, although it may be difficult to decide what this mapping should be. We are not concerned with the problems of deciding on a mapping, rather on what should be loaded, and how it can be executed.

In particular we are concerned with what should happen to the "main body" of the program. For the sake of simplicity let us assume that the main body of the program consists of a sequence of declarations of tasks which are to be run on a number of separate machines. Clearly the code for the individual tasks should be loaded on the designated machines together with code for instantiating the tasks. Address information to allow inter-task communication must also be made available to the kernel (this may be regarded as the vestige of the declarations of the other tasks). We can rely on the semantics of the rendezvous to ensure correct behaviour despite the fact that tasks which wish to communicate may be created at significantly different times.

The above solution is satisfactory so long as no task tries to create a task to run in another machine. If tasks can be created in other machines, then a mechanism has to be provided for one kernel to request another to create and run a task. This facility may cause problems if the processes in separate machines wish to share data, and it may also lead to difficulties in scheduling, and in assessing the amount of mill time absorbed by any task, etc. Similar comments apply to the inclusion of executable code in the main body of the program.

It would seem to be by far the simplest if Ada programs to run in MCS were restricted in the following ways. First, the main body may only consist of the declaration of tasks. Second, no task may create a task to run on another machine. It seems likely that these restrictions would be acceptable in practice.

10) Conclusions

We have described some possible ways in which an Ada rendezvous could be implemented in an MCS. We have also considered the overheads of using the rendezvous where we wish to perform 1:N, rather than 1:1, communication. We have shown that the overheads of using the Ada rendezvous as opposed to a message based communication system are quite large. This is, perhaps, not surprising as procedure calls are fundamental within a single computer, but message passing, rather than remote procedure calls, are fundamental to communication systems.

We have briefly considered the problem of loading and executing Ada programs, and we have suggested a rule for constructing and mapping Ada programs which would simplify their implementation on an MCS.

We have not covered all the important issues to do with implementing Ada on an MCS. For example we have ignored problems of deciding on a good software to hardware mapping; how we test and monitor program execution; how we extend or replace parts of the running program etc. This omission can be justified by saying that the other problems are pertinent to other programming languages, not just Ada. What we have tried to do is to concentrate on those problems which seem to be peculiar to Ada.


ATE
LME